MeshTaichi: A Compiler for Efficient Mesh-based Operations

CHANG YU*, University of Electronic Science and Technology of China and Taichi Graphics, China
YI XU*, Tsinghua University and Taichi Graphics, China
YE KUANG, YUANMING HU, and TIANTIAN LIU, Taichi Graphics, China

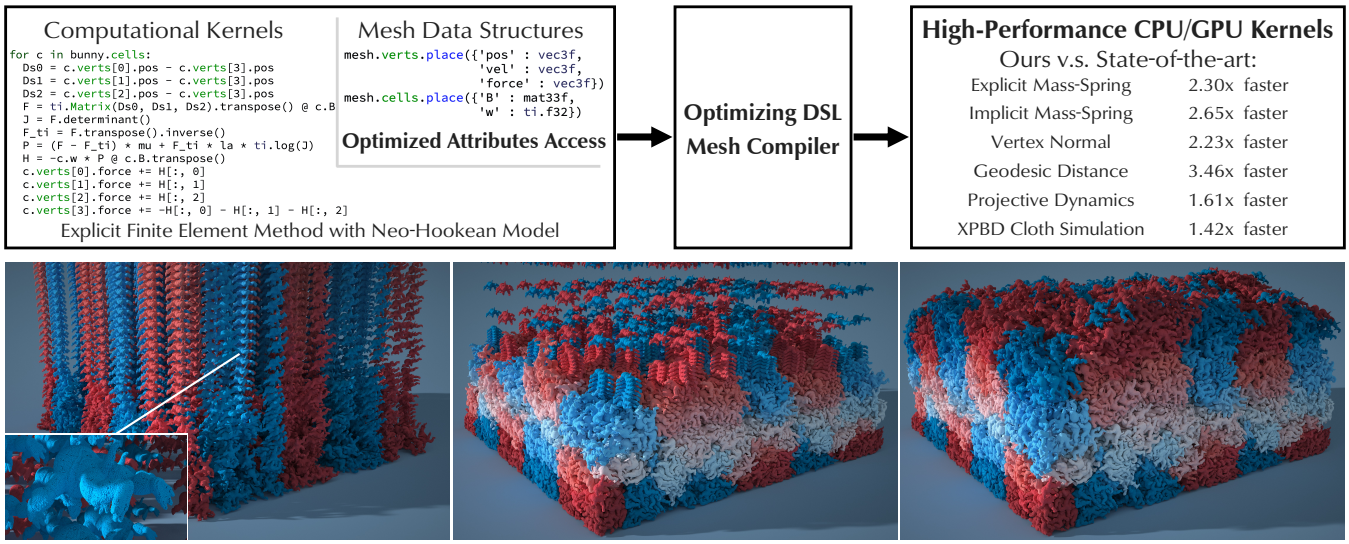


Fig. 1. **Top:** We propose MeshTaichi: a high-level programming model to handle mesh-based operations efficiently. By exploiting data locality and memory hierarchy, our compiler generates high-performance CPU/GPU computational kernels. As a result, we achieve much better performance than the state-of-the-art compilers and data structures. **Bottom:** We implement the MLS-MPM algorithm with FEM-based Lagrangian-force using our programming language. The simulation consists of 17, 010 armadillos with 222, 048, 540 vertices and 713, 739, 600 tetrahedrons in total. Each frame consists of 300 substeps and takes 2.4 minutes on average on an NVIDIA A100 Tensor Core GPU.

Meshes are an indispensable representation in many graphics applications because they provide conformal spatial discretizations. However, mesh-based operations are often slow due to unstructured memory access patterns. We propose MeshTaichi, a novel mesh compiler that provides an intuitive programming model for efficient mesh-based operations. Our programming model hides the complex indexing system from users and allows users to write mesh-based operations using reference-style neighborhood queries. Our compiler achieves its high performance by exploiting data locality. We partition input meshes and prepare the wanted relations by inspecting users' code during compile time. During run time, we further utilize on-chip memory (shared memory on GPU and L1 cache on CPU) to access the wanted attributes of mesh elements efficiently. Our compiler decouples low-level optimization options with computations, so that users can explore different localized data attributes and different memory orderings without changing their computation code. As a result, users can write concise code using our programming model to generate efficient mesh-based computations on both CPU and GPU backends. We test MeshTaichi on a variety of physically-based simulation and geometry processing applications with both triangle and tetrahedron meshes. MeshTaichi achieves a consistent speedup ranging from 1.4× to 6×, compared to state-of-the-art mesh data structures and compilers.

CCS Concepts: • **Software and its engineering** → **Compilers**; • **Computing methodologies** → **Parallel programming languages**; **Physical simulation**; **Mesh models**.

Additional Key Words and Phrases: Domain-specific languages, GPU computing, Mesh, Physically-based Simulation.

ACM Reference Format:
Chang Yu, Yi Xu, Ye Kuang, Yuanming Hu, and Tiantian Liu. 2022. MeshTaichi: A Compiler for Efficient Mesh-based Operations. *ACM Trans. Graph.* 41, 6, Article 252 (December 2022), 18 pages. https://doi.org/10.1145/3550454.3555430

*Both authors contributed equally to this research.

Authors' addresses: Chang Yu, University of Electronic Science and Technology of China, Taichi Graphics, Beijing, China, g1n0st@live.com; Yi Xu, Tsinghua University, Taichi Graphics, Beijing, China, BillXu2000@outlook.com; Ye Kuang, ykuang.me@gmail.com; Yuanming Hu, yuanmhu@gmail.com; Tiantian Liu, Taichi Graphics, Beijing, China, ltt1598@gmail.com.

0730-0301/2022/12-ART252 $15.00
https://doi.org/10.1145/3550454.3555430

1 INTRODUCTION

Meshes are widely used as discretization forms in many physically-based simulation and geometry processing applications. Unlike regular grid data structures, a mesh can conform to any desired geometry with localized resolutions. However, the advantage of meshes comes at a cost. First, the topological information of meshes need to be managed explicitly. One can compute the indices of a grid element's neighbors by simply offsetting its grid index; however, they need to lookup a relation table to decide where a mesh element's neighbor are. It can be worse when the wanted relation is not stored explicitly in the relation tables so it needs to be computed on the fly. For example, to find all neighboring vertices of a certain vertex, we often need to first check the edges connected to that vertex, and then acquire the other vertices of those edges. Second, even if the indices are given, the access of these mesh attributes is often in a less coherent way. Although mesh-based operations are usually local, the data attributes associated with these operations can be scattered far away in memory. For example, when we query the position attributes of three vertices of a triangle, the indices of these vertices are not very likely to stay together. The attributes of these three vertices may not exhibit spatial locality. The situation can be even worse in GPUs where memory access is more expensive than computations. In short, compared to grid-based operations, the major overhead for mesh-based operations comes from two aspects: (1) we need **extra relation data structures** to maintain the indices of wanted attributes in a mesh; (2) we often query these mesh attributes via unstructured access patterns that lead to **more frequent cache misses**.

Improving performance for mesh-based operations is not an easy task for users. It requires repetitive profiling and tuning to optimize the memory layout and the caching scheme. While the shared memory on GPU and the L1 cache on CPU provide significant speedup at hardware level, manually utilizing these memory hierarchies often fails to achieve high performance and high productivity at the same time – it usually ends up with either inefficient or lengthy code. Producing efficient and easy-to-use mesh-based operations is a challenging problem for high-performance mesh-based applications. To be more specific, we desire a concise programming interface to hide the complex indexing problem for all types of meshes, and simultaneously, we strive for high performance via low-level data access optimizations to fully utilize powerful on-chip memory. Therefore, we believe a high-level mesh programming model and a corresponding high-performance mesh compiler shall be the key.

We propose MeshTaichi, a novel mesh compiler that provides an intuitive programming model to handle mesh-based operations efficiently. We summarize our main design goals and corresponding contributions as follows:

1. **Uniformly handle all kinds of mesh elements in both 2D and 3D**, including vertices, edges, triangle faces, and tetrahedron elements. Unlike regular grids, meshes are naturally versatile and involve much more complex topology. We design an intuitive programming model that enables users to access all kinds of elements. We support *natural-language-like grammar* to queries different mesh elements such as "for every vertex in a bunny".
2. **Intuitive syntax for neighborhood access.** While meshes involve complex relations, we want to hide all the complexity from programmers. Instead of tracing all elements using their global indices indicating physical memory addresses, our system allows users to query attributes of neighborhood elements using *reference styles*, such as "velocity of the fourth vertex in a tetrahedron" or "positions of all neighboring vertices of a vertex". We achieve this by partitioning input meshes into patches and pre-computing all wanted relations for each patch by inspecting users' code in compile time.
3. **Efficient mesh attribute access.** The unstructured memory access pattern for mesh attributes is a major problem to slow mesh-based operations down. Our compiler *takes over the caching scheme* for mesh attributes, managing the attribute exchange between global and shared memory on our own. This ensures most mesh attributes are cached before they are needed for computations, greatly reducing memory overhead for attribute reads and writes.
4. **Decoupled low-level mesh optimization options from implementation**. Our compiler also decouples the low-level optimization from computations. Programmers can query any attribute of any mesh element without knowing which patch that element may belong to or what memory ordering that attribute is saved with. We further provide some *compiler optimization hints*, allowing users to exploit different cached attributes and different memory orderings without changing their computation code.
5. **Automatic parallelized execution for multiple backends.** Modern simulations often need to handle large meshes. Our compiler has a solid backend implementation that automatically generates efficient kernels to compute on mesh attributes within on-chip memory in parallel, for both CPU and GPU architectures. We also reduce the inter-patch communication overhead by investigating patching algorithms.

We conduct a thorough evaluation of our system, demonstrating its performance and scalability. Our system provides a consistent speedup ranging from 1.4× to 6× compared with other mesh data structures and compilers. We also provide several state-of-the-art implementation of mesh-based simulations and geometry processing algorithms as by-products. We build our mesh compiler upon the open-source programming language Taichi [Hu et al. 2019]. The *source code* of our implementation is now officially a part of Taichi.

2 RELATED WORK

Meshes naturally provide conforming discretizations to wanted geometries. They are frequently used in computer graphics applications to represent the virtual objects such as cloth [Baraff and Witkin 1998; Terzopoulos et al. 1987], hair [Yuksel et al. 2009], plants [Zhao and Barbič 2013], flesh [Kadleček et al. 2016], faces [Blanz and Vetter 1999] or even humans [Chen et al. 2021]. Many research works focus on computing the quantities on meshes like the vertex normal [Max 1999], the geodesic distance [Calla et al. 2019], and the mean curvature flow [Desbrun et al. 1999], whereas some other works use the explicit topological information of meshes to accelerate their applications [Liu et al. 2013; Narain et al. 2016; Xian et al.

2019]. Meshes can also be used together with structured grids to produce interpenetration-free results [Fang et al. 2019; Jiang et al. 2017]. There are applications (e.g., rendering) requiring only the surface structure and applications taking the volumetric information of meshes. Tools such as Tetgen [Si 2015] and TetWild [Hu et al. 2018b] are used to convert surface meshes to their corresponding volumetric representations.

Unlike grid-based structures where users can implicitly infer any grid attribute's data address, managing the data attributes and relations on meshes is more complicated due to their unstructured typologies. Varies of data structures are used to access the mesh data efficiently. Quad edges [Guibas and Stolfi 1985] focus on data structures for Voronoi diagrams, supporting users to visit the dual and the mirror-image information easily. Winged edges [Baumgart 1972] support polygonal faces. The winged-edge data structure stores neighboring faces, edges, and vertices for each edge. Half edges [Mäntylä 1987] are designed for two-dimensional polygonal surface meshes. It divides one edge into two halves to represent the two neighboring faces and allows users to visit the next, the previous, and the opposite counterpart of the edges. Directed edges [Campagna et al. 1998] provide a special treatment for triangle meshes to improve the performance further. These data structures achieve constant time complexity to perform neighbor queries, but they are not optimized for parallel computing architectures. RXMesh [Mahmoud et al. 2021] introduces a mesh data structure specified for GPU. It subdivides the mesh into patches and performs better by caching the mesh relations into the shared memory. While mesh data structures provide a reliable attribute accessing scheme for users, their static memory allocation brings redundant costs when users only want to visit a small fraction of the mesh (e.g., only the vertices and edges but not the faces).

Many libraries provide canned implementations to save users' efforts. OpenMesh [Botsch et al. 2002] and CGAL [Kettner and Cacciola 2006] provide a stable and solid implementation of the half-edges. RXMesh [Mahmoud et al. 2021] provides an accompanying implementation to its own data structure with a further improvement utilizing the shared memory. The disadvantage of the libraries come from two aspects. First, most libraries are designed upon a specific programming language, such as C++ or CUDA. The underlying languages usually restrict the portability of these libraries. Second, the libraries miss the chance to optimize users' code in compile-time; hence they need to expose many low-level interfaces to users as further acceleration hints.

Some previous works resort to domain-specific languages (DSLs) to overcome the portability and productivity problems aforementioned. GraphIt [Zhang et al. 2018] is a DSL for graph computations on CPU. GraphIt decouples algorithm from optimization for graph applications. Users can use GraphIt's optimization representation to easily switch to different data layouts and try different combinations of optimization techniques. But GraphIt is not optimized for GPU applications. The unified GraphIt compiler(UGC) [Brahmakshatriya et al. 2021] extends GraphIt to multiple backends including GPU. But it only focuses on proper scheduling to improve load-balancing. Both GraphIt and UGC do not manage the caching scheme explicitly hence miss the opportunity to exploit data locality. Simit [Kjolstad et al. 2016] allows the users to view the data relations as a form of hyper-graph, where all the computations are applied on the abstracted vectors, matrices, and tensors. The abstraction helps users focus more on the computation than on the low-level attribute-access details. Liszt [DeVito et al. 2011] is a DSL for constructing mesh-based PDE solvers. The focus of Liszt is to provide portable programming models with heterogeneous processors and hardware accelerators. Ebb [Bernstein et al. 2016] is a new implementation of the Liszt programming model designed to support a broader range of simulation applications other than pure PDE solvers. Ebb uses a three-layer architecture to separate the frontend code, the domain libraries, and the run-time libraries. The domain library layer of Ebb is programmable, supporting a wider range of geometric domains. Both Simit and Ebb handle the input mesh as a general formed matrix or graph. These high-level linguistic abstractions provide portable and productive programming models, but they do not further exploit data locality to improve the performance. Detailed discussions about why we need new programming languages and about the core design strategies of Simit and Ebb can be found in Bernstein and Kjolstad [2016].

Data locality is a crucial factor for large-scale computations. One way to improve the data locality is to manage the order and layout of the stored data manually, such as using Morton code to store particles using a Z-order curve in particle-based fluid simulations [Ihmsen et al. 2011]. Another way is to subdivide data into smaller chunks and to perform computations chunk-wisely. For instance, Gao et al. [2018] apply a sparse grid with scratchpad optimizations to utilize the shared memory for the material point method. Ghost cell patterns [Kjolstad and Snir 2010] can be used to provide cached data at the periphery of chunks for regular grids. RXMesh [Mahmoud et al. 2021] further extends the idea of ghost-cells to irregular meshes, and calls them the "ribbons".

Taichi [Hu et al. 2019] is an open-source data-oriented DSL which improves data locality using compiler knowledge. Taichi has a unique structural node system that decouples the computation from the underlying data layouts. This decoupling enables users to switch among different memory layouts without changing the computation. However, Taichi only exploits data locality for structured grids but not for meshes. Users need to implement all the mesh data structures from scratch and manage the indexing systems on their own. We build our programming language upon Taichi to inherit its portability. We further design our novel mesh data localization scheme to exploit data locality for efficient mesh-based operations.

Relation to RXMesh. Our mesh compiler has a conceptual affinity to the state-of-the-art mesh library RXMesh because we both subdivide a mesh into smaller patches and we both use ribbons to reduce the inter-patch communication overhead. The fundamental design divergence between our system and RXMesh comes from our **data localization strategies**: our compiler localizes mesh *attributes* only while RXMesh localizes mesh *relations* only. As a library, RXMesh can not analyze wanted relations and attributes from users' code before run time automatically. It therefore adopts a compact linear algebraic representation [DiCarlo et al. 2014] to represent mesh relations and caches these relations into shared memory so users can compute the indices of their wanted relation efficiently in run

time. RXMesh does not cache any mesh attributes which are usually not stored coherently (such as vertex normals or edge lengths). It still suffers from frequent cache misses. We observe that mesh-based operations are often called repeatedly from applications, e.g. the spring force evaluation can be called hundreds times within a second in a mass-spring simulation. Computing relations every time can be a waste. As a compiler, our system can inspect users' code to pre-compute all required relations from certain applications and store those relations in global memory during compile time. In run time, our compiler manages the caching scheme for mesh attributes explicitly to exploit data locality. We take over the attribute data exchange between shared and global memory to ensure that most attributes are cached before they are needed for computations. To balance between data locality and occupancy, we choose *not* to cache any relations to maximize performance. We conduct thorough validations for this design decision in Section 7.2. Besides this major difference on data localization, our system also provides a more **concise programming interface**, a more flexible programming model that **supports multiple relations** inside one computation kernel, and more expressive relation coverage including **volumetric elements and edge-edge relations**, compared to RXMesh.

3 BACKGROUND

3.1 Meshes, Relations, and Attributes

We define a triangle mesh using $\mathcal{M}_3 = \{\mathcal{V}, \mathcal{E}, \mathcal{F}\}$ and define a tetrahedron mesh using $\mathcal{M}_4 = \{\mathcal{V}, \mathcal{E}, \mathcal{F}, \mathcal{C}\}$, consisting of vertices $\mathcal{V}$, edges $\mathcal{E}$, faces $\mathcal{F}$ and cells $\mathcal{C}$. We call a vertex, an edge, a face or a cell a *mesh element*, or an element for short. A n-d element represents an element consisting of $n+1$ vertices. For example, vertices are 0-d elements, edges, faces, and cells are 1-d, 2-d, and 3-d elements, respectively. Two n-d elements are neighbors *iff* they share a $(n-1)$-d element. The $\mathcal{VV}$ neighborhood, as a special case, is defined when two vertices are in the same edge since vertices are already 0-d elements. A n-d element and a m-d element ($n > m$) are neighbors *iff* the m-d element is exactly a subset of the n-d element. Taking a mass-spring system as an example, two springs (two 1-d elements) are neighbors *iff* they share one vertex; A spring's 0-d neighbors are its two end-points; A vertex's 1-d neighbors are all the springs connected to it.

A *relation* $\mathcal{AB}$ defines the neighborhood between two *mesh elements* $\mathcal{A}$ and $\mathcal{B}$, as we always access the relation from $\mathcal{A}$ to $\mathcal{B}$, we call $\mathcal{A}$ the *from-end* element and call $\mathcal{B}$ the *to-end* element. If $\mathcal{A}$ is an n-d element and $\mathcal{B}$ is an m-d element in relation $\mathcal{AB}$, we call the relation a *static relation* when $n > m$ because each *from-end* element has a static number of *to-end* neighbors. Otherwise, we call the relation a *dynamic relation*, because each *from-end* element may have different numbers of *to-end* neighbors. For instance, a $\mathcal{CV}$ relation is an *static relation* since each tetrahedral cell has four vertex neighbors, and a $\mathcal{VE}$ relation is an *dynamic relation* because a vertex can be linked with arbitrary number of edges.

We refer *mesh attributes* (or attributes for simplicity) as the quantities defined on certain types of elements. For example, in a mass-spring system, each vertex may have its *mass*, *position*, *velocity*, and *force* attributes, and each edge has a *rest_length* attribute.

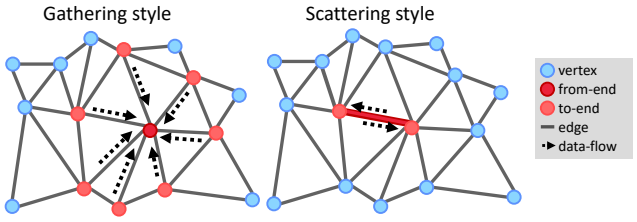


Fig. 2. Two typical styles of local mesh-based operations. **Left:** The *gathering-style* operation in a $\mathcal{VV}$ relation. **Right:** The *scattering-style* operation in an $\mathcal{EV}$ relation.

3.2 Mesh-based Operations

We focus on local mesh-based operations which can load and store attributes from arbitrary elements and their neighbors. While mesh-based operations may involve different attributes, they have two typical styles: *gathering* style and *scattering* style. The *gathering*-style operations read attributes from the neighbors of an element and store attributes to the element itself. The *scattering*-style operations, on the contrary, reads attributes from an element and writes attributes to its neighbors. Note that users can perform different mesh operations to achieve the same goal. For example, in a mass-spring system where users want to compute the force for each vertex, they can either use the $\mathcal{EV}$ relation to store the force attributes using the *scattering*-style operations, or use the $\mathcal{VV}$ relation to compute the force attributes using *gathering*-style operations as shown in Figure 2.

3.3 Data flow of a typical Mesh-based Operation

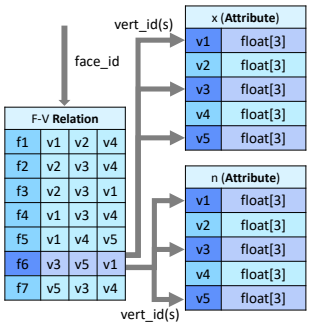


Fig. 3. Typical data flow to perform vertex normal computations on a mesh.

Compared with grid-based operations, mesh-based operations have much higher memory access overhead. This is caused by the unstructured memory access pattern from meshes. Let us take a vertex normal computation as an example to illustrate the data flow of a typical mesh-based operation and to explain why such mesh-based operations are slow. In this example, we need to first compute the surface normal for each triangle using positions of its vertices first, then scatter the computed surface normal to its vertices. As we can see from Figure 3, we need to first visit an $\mathcal{FV}$ relation table to load the indices of vertices belonging to a face and visit a position array using the indices to load the position attributes for these vertices. Once the face normal is computed, we need to visit another normal array to store the normal back. This simple mesh operation involves at least one *relation* access and two *attribute* accesses for every triangle face. The cache hit rate can make it even worse: due to the nature of unstructured memory access for meshes, the wanted *attributes* are less likely to stay contiguously in

global memory. The storing statement can further hurt the performance because it raises potential data races and is often protected by expensive atomic operations.

4 PROGRAMMING MODEL

We demonstrate the usage of MeshTaichi using an explicit finite element method example with the Neo-Hookean model following the course note [Sifakis and Barbic 2012]. The First Piola-Kirchhoff stress tensor is given as:

$$P(F) = \mu(F - F^{-T}) + \lambda \log(J) F^{-T} \tag{1}$$

where F describes the deformation gradient, J is the determinant of F, and μ and λ are the Lamé coefficients.

4.1 Describing the Mesh Data

We need to define mesh data structures before applying them to computations. In this section, we provide the following APIs to create our new mesh data types:

```
mesh = ti.TetMesh() # new tetrahedron mesh data type
# mesh = ti.TriMesh() # new triangle mesh data type
```

Note that `mesh` is a mesh data type instead of a mesh instance. A `mesh` can be instantiated by multiple models.

The next step is to define the attributes for each mesh element. The type of a mesh attribute can be a scalar, a vector, a matrix, or their corresponding quantized versions [Hu et al. 2021]. In this FEM example, each vertex has three attributes: the position `pos`, the velocity `vel` and the force applied to it `force`. All three attributes are floating-point type three-dimensional vectors based on the Taichi type system. Each (tetrahedral-)cell in the mesh has two attributes: the rest-pose volume `w` which is a floating-point scalar, and the inverted rest-pose shape matrix `B` which is a 3×3 matrix. The code snippet defining the mesh elements is listed below.

```
mesh.verts.place({'pos' : ti.math.vec3,
                  'vel' : ti.math.vec3,
                  'force' : ti.math.vec3})
mesh.cells.place({'B' : ti.math.mat3,
                  'w' : ti.f32})
```

Once the data types are all defined, users can instantiate a model with an external file:

```
bunny = mesh.build('./bunny.mesh')
```

Our compiler can instantiate triangle meshes using surface representation formats like ".obj" and ".ply" files, or volumetric representation formats like ".mesh" file. Once the mesh model is instantiated, our compiler will further partition the meshes to generate metadata. Details are further discussed in Section 6.4.

4.2 Computing on a Mesh

The mesh computations are declared within a `mesh-for` of a kernel, which can be as simple as shown below.

```
# parallel loop over all mesh cells
for c in bunny.cells:
  ...
# parallel loop over all mesh vertices
for v in bunny.verts:
  ...
```

A `mesh-for` has very similar syntax as a `range-for` which loops over the indices in an interval. It is specified with a mesh object's element type (e.g., cells or vertices). Each element inside a mesh is queried with an index-free reference style. The elements' indices and their corresponding memory addresses are hidden from the users. Our compiler parallelizes the outermost `mesh-for` loops with high-performance kernels, so that the looped mesh elements are efficiently computed in parallel.

Most mesh-based operations involve not only the attributes of an element, but the attributes of the neighboring elements as well. Our compiler allows users to access neighbor attributes either with reference-based queries using another nested sequential `mesh-for` loop or with index-based queries using a `range-for` loop as shown below.

```
for c in bunny.cells:
  total_force = ti.math.vec3(0)
  # reference-based access
  for v in c.verts:
    total_force += v.force
```

```
for c in bunny.cells:
  total_force = ti.math.vec3(0)
  # index-based access
  for i in range(c.verts.size):
    total_force += c.verts[i].force
```

Now we have the way to access the neighbors. Let us wrap everything up and write a `substep()` function to calculate the force (according to Eq. 1) and perform explicit time integration in a finite element simulation:

```
@ti.kernel
def substep():
  for c in bunny.cells:
    Ds0 = c.verts[0].pos - c.verts[3].pos
    Ds1 = c.verts[1].pos - c.verts[3].pos
    Ds2 = c.verts[2].pos - c.verts[3].pos
    F = ti.Matrix(Ds0, Ds1, Ds2).transpose() @ c.B
    J = F.determinant()
    F_ti = F.transpose().inverse()
    P = (F - F_ti) * mu + F_ti * la * ti.log(J)
    H = -c.w * P @ c.B.transpose()
    c.verts[0].force += H[:, 0]
    c.verts[1].force += H[:, 1]
    c.verts[2].force += H[:, 2]
    c.verts[3].force += -H[:, 0] - H[:, 1] - H[:, 2]

  for v in bunny.verts:
    # assuming unit mass for simplicity
    v.vel += dt * v.force
    v.pos += dt * v.vel
```

Our language frees users' burden to think about using atomic operations. The `+=` operators inside a `mesh-for` will be atomic by default to avoid race conditions. Our compiler will analyze the code and demote the unnecessary atomic operations automatically.

4.3 Interacting with Non-mesh Data

Recall that we visit all elements in a `mesh-for` using their references, users do not need to bookkeep their corresponding indices. However, there are cases where the indices of elements are wanted. In these cases, we refer users to visit the `id` attribute of an element. We demonstrate an example to export the positions of vertices to an external multi-dimensional array as follows:

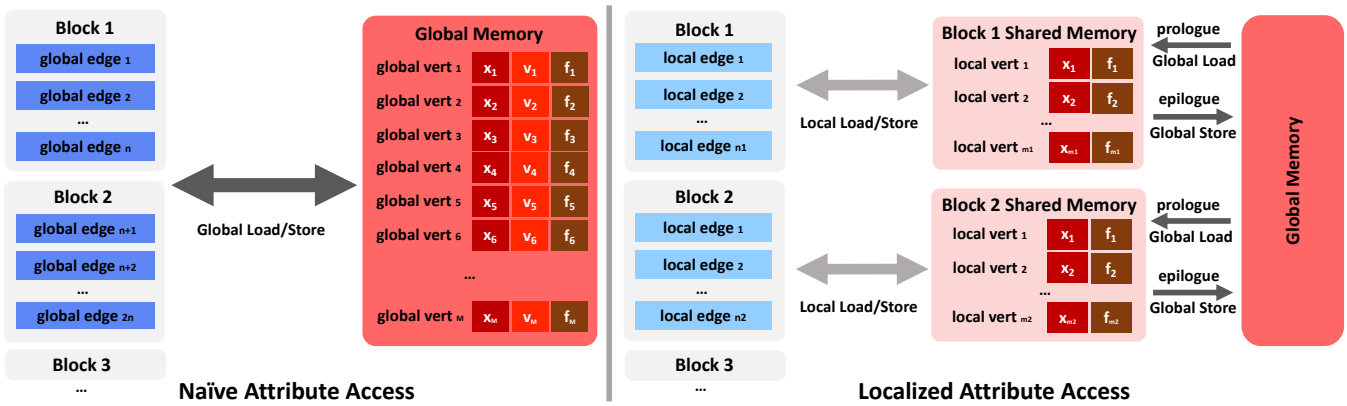


Fig. 4. **Left:** Naïve data access for a physically-based simulation. The program visits the global memory directly to access the position (x), velocity (v) and force (f) data. **Right:** Our compiler localizes the position (x) and force (f) data into shared memory in this example. Visiting each cached attribute only requires a shared memory access using its *local index*. We load the batched attribute data from global memory during a *prologue* pass and write them back during an *epilogue* pass.

```
# An 1D array of len(bunny.verts) 3D f32 vectors
pos_ex = ti.Vector.field(3, ti.f32,
    shape=len(bunny.verts))

@ti.kernel
def export():
  for v in bunny.verts:
    pos_ex[v.id] = v.pos
```

5 MESH-BASED COMPILER DESIGN

We observe that mesh-based operations are mostly *out-of-cache* operations, making mesh-based applications memory bound and low computational throughput. To be more specific, we summarize the reasons for our observation as follows:

(1) In the modern hardware memory hierarchy, a global memory (main memory in CPU and device memory in GPU) access is often 10 $\sim$ 100$\times$ slower than on-chip memory access. The irregular structure of a mesh makes it more likely to miss the cache when fetching mesh attributes.
(2) GPUs require coalesced global memory access to maximize bandwidth. The mesh computations with frequent neighbor attribute visits incur irregular memory accesses, which are hard to coalesce. The performance is often bound by such access patterns.
(3) Some mesh-based operations compute attributes on an element and store those attributes to that element's neighbors. We need to consider expensive atomic instructions to avoid any data race in these *scattering*-style operations.
(4) Some queries involve *dynamic relations* (e.g. a $\mathcal{VV}$ relation). These queries can visit different numbers of *to-end* neighbors for different *from-end* elements. This raises potential branch divergence problems that can further cause more cache misses.

Naïve implementations of mesh-based operations involve frequent global memory access in an unstructured way, which is apparently inefficient as illustrated in the left of Figure 4. To perform a mesh kernel efficiently, an ideal implementation would break the massive mass-based operations into smaller blocks, execute the computation of these small blocks independently, and even better, cache all the computation-related mesh attributes into shared memory in advance, as shown in the right of Figure 4. We target efficient memory access as the top priority for our mesh compiler to address the issues aforementioned. Therefore, our compiler manages the attribute data exchange between global and shared memory for mesh-based operations.

Our system partitions input meshes into smaller patches once they are loaded. We then inspect the `mesh-for` kernels to decide the wanted *relations* for each patch. We compute those relations and store them into global memory. The mesh partitioning and relation preparation steps are done in compile time. We also save these information to files as our compiler metadata to prevent unnecessary re-computations.

During run-time, our compiler loads all wanted mesh *attributes* into shared memory in a *prologue* stage before the computations take place, and stores necessary *attributes* back to global memory in an *epilogue* stage when the computations are done. The *attribute* localization strategy naturally addresses issue (1) since local memory access is orders of magnitudes faster than global access. It will also mitigate issue (3) and (4), because we reduce race conditions by eliminating most inter-patch conflicts, and our cache misses due to branch divergence is less expensive since *dynamic relations* will only visit different numbers of local memory addresses instead of global ones.

With our attribute localization scheme, our global memory access happens mostly in *prologue* and *epilogue* stages. We also want these memory access to be coalesced to address issue (2). Our compiler further manages memory ordering and provides users with different ordering options for all mesh attributes. Our design decouples the ordering from computations so that users can explore different ordering strategies without changing the computation kernels.

5.1 Mesh Partitioning and Relation Preparation

We want to cache mesh attributes into shared memory. But apparently the size of shared memory is too small to fit large mesh data as a whole. We, therefore, need to partition input meshes into smaller patches. The key idea of patching is to cluster mesh elements so that each element can access its own attributes and its neighbors' attributes in the same block of shared memory. To accelerate the neighbor-attribute-accessing for peripheral elements inside a patch, we pad an extra layer of elements from neighboring patches, similarly with the ghost cell pattern [Kjolstad and Snir 2010] and ribbons in RXMesh [Mahmoud et al. 2021]. We also adopt the name of *ribbons* from RXMesh for these padded elements and call them *ribbon elements*. We refer other elements inside a patch as *owned elements*.

To maximize the utilization of shared memory, we pose two objectives for our patching algorithm. First, the size of each patch needs to be bounded, so we can estimate the shared memory size for each patch easily. It would be even better if the sizes of all patches can reach their upper bound to optimize load balancing. Second, we want as fewer ribbon elements as possible because the attributes of these ribbon elements are shared by at least two patches so that we need to duplicate these attributes in shared memory for different patches. For a given mesh, the total number of attributes fetched from global memory, including the duplicated ones for those ribbon elements, is proportional to $\gamma = \frac{n_r+n_o}{n_o}$, where n_o is the total number of *owned elements* that only depends on the input mesh, and n_r is the total number of *ribbon elements* that also depends on the patching algorithm. We use γ as the normalized theoretical global memory access to indicate performance. Higher γ leads to more total attributes fetched from global memory which indicates lower performance. That is why we want to reduce the number of ribbon elements by picking a good patching algorithm.

Several approximated algorithms can be applied to tackle this patching problem. RXMesh uses a modified k-means clustering to partition their meshes. Since k-means can not control the cluster size explicitly, some patches may exceed their size limit. Extra seeds need to be placed to further break those oversized patches. The problem of k-means comes from its lack of patch size control. K-means tends to partition input meshes into smaller-than-expected patches to satisfy patch size constraints. The problem can be worse when partitioning tetrahedral meshes where degrees of each elements increase severely in three-dimensional space. We can for sure pack small patches together to improve load balancing. But smaller patches also end up with more ribbon elements which is a harder problem to fix.

We observe that the patch size is a crucial indicator for a good partitioning. Larger patch sizes indicate that: first, patches are filled with as many elements as possible, and second, patches tend to have fewer ribbon elements – assuming that patches are approximately shaped as convex. We therefore apply a greedy algorithm to maximize the size for each patch directly as shown in Alg. 1. However, one disadvantage of this greedy patching algorithm is that it generates a few tiny patches. Those tiny patches are often results from some orphan elements whose neighbors are occupied by other large patches. To improve load balancing, we further pack these tiny patches with other patches until reaching their size limit. We also perform the same packing strategy for k-means and find

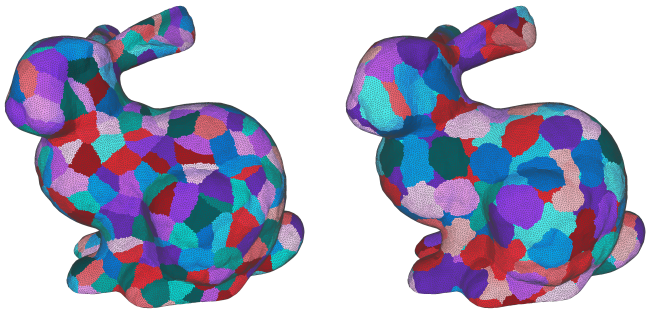


Fig. 5. Patches generated using: **left:** k-means algorithm, and **right:** our greedy algorithm.

that the total number of patches generated by our greedy algorithm and k-means are comparable in most cases. However, the our greedy patching algorithm makes the majority of elements stay in notably larger patches compared to k-means, as shown in Figure 5. As a result, the greedy patching algorithm also reduces the ribbon elements in most of our testing cases, especially for three-dimensional examples. We analyze the influence of patching algorithms in the validation section in Section 7.1.

ALGORITHM 1: Greedy Patching Algorithm.

```
while not all elements are patched do
    find an unpatched element u
    S ← {u}
    while |S| < patch size do
        f(v) ← sum(connected(v, w) for w in S)
        v = argmax(f(v); v is not in any patch and not in S)
        S ← S ∪ {v}
    end
    save S as a new patch
end
```

The localized patches generates another indexing system. We refer the original index of each element as the *global index* which points to global memory address. After the mesh partitioning, each element will be owned by one patch and be labeled with a unique index to target within the patch, accessing (local) shared memory. We call this index inside each patch a *local index*. Once the mesh partitions are made, we prepare the wanted relations for each patch by inspecting `mesh-for` kernels. We store the relations using their local indices only, and setup a local-to-global mapping for each mesh element to access its global index. Since the range of local indices is limited by patch sizes, we use two-byte unsigned integers to represent the local indices. This further reduces the relation storage by half for large meshes. The estimated memory footprint for relations is reported in Appendix B.

Both mesh partitioning and relation preparation steps are done in compile time. Our compiler also file-caches these information as compiler metadata. These metadata will be re-computed for changed meshes.

5.2 Mesh Attribute Localization

As illustrated in Figure 3, a simple mesh-based operation may involve multiple global memory accesses to query the relation table and to load/store attributes. Shall we cache all of them into shared memory? Our answer is no. The size of shared memory is limited, higher consumption of shared memory often ends up with lower occupancy. So we need to be very careful about shared memory usage. We choose to localize only the wanted mesh *attributes* into shared memory and leave the pre-computed *relations* in global memory. Because the relations have been already computed in compile-time, the relation queries in our system are relatively structured, even for *dynamic relations*. Detailed validation experiments can be found in Section 7.2.

5.2.1 Localizing Mesh Attributes with Optimization Hints. Our compiler provides a hint syntax `mesh_local` as a decorator of a `mesh-for` to localize the mesh attributes by need:

```
1 # scheme 1: put attributes pos and force of vertices
      into the shared memory
2 ti.mesh_local(bunny.verts.pos, bunny.verts.force)
3 for c in bunny.cells:
4     ...
```

The vertices attributes `pos` and `force` are the position and force defined for the finite element method in Section 4.1. By inserting this hint before line 3 (`for c in bunny.cells:`) defined in Section 4.2, our compiler caches the `pos` and `force` into the shared memory and replaces the attribute access from global memory to the local memory without modifying the computation code.

Our compiler executes a systematic analysis and transform the index space for these mesh attributes with this hint. We divide the transformation process into three passes. First, the compiler emits a *prologue* pass to fetch the wanted attributes from global memory to shared memory. Second, as the attributes have been localized, we can access and update them from shared memory based on their local indices. At last, when attributes are updated due to write-access, our compiler will emit an *epilogue* pass to write mesh attributes back to global memory as shown in the right of Figure 4. Since we write data back to global memory in batches, potential write-conflicts encountered by our *epilogue* pass is greatly reduced compared to an unoptimized attribute access scheme. We will discuss the implementation of transformation passes in detail in Section 6.2.

Remember that there is a trade-off between shared memory usage per block and occupancy in the GPU? When shared memory size per block increases, the occupancy will decrease because a multiprocessor's shared memory is partitioned among the resident thread blocks. Moreover, the shared memory size is limited and will be quickly exhausted if we cache too many attributes. We therefore *decouple* our hint mechanism from the computation code, allowing the developers to exploit the best attribute caching scheme with little effort. For example, the aforementioned scheme can be easily switched to other schemes:

```
1 # Scheme 2: only cache attributes force
2 ti.mesh_local(bunny.verts.force)
3 for c in bunny.cells:
4     ...
5 # Scheme 3: cache attributes pos in vertices and B in
      cells
6 ti.mesh_local(bunny.verts.pos, bunny.cells.B)
7 for c in bunny.cells:
8     ...
```

Complex mesh-based operations may involve multiple mesh attributes and external arrays. Our `mesh_local` decorator provides an uniform high-level hint syntax for them as well:

```
1  mesh.verts.place({'x' : ti.f32})
2  mesh.edges.place({'y' : ti.f32})
3  mesh.cells.place({'z' : ti.f32})
4  bunny = mesh.build("./bunny.mesh")
5  w = ti.field(type=ti.f32,
6           shape=len(bunny.verts)) # external field
7  @ti.kernel
8  def foo():
9      # cache both mesh attributes and an external field
10     ti.mesh_local(bunny.edges.y, bunny.verts.x, w)
11     for c in bunny.cells:
12         for v in c.verts: v.x += c.z # C-V
13         for e in c.edges: e.y += c.z # C-E
14         for e in c.edges:
15             for v in e.verts: w[v.id] += c.z # C-E-V
```

5.2.2 Localizing Mesh Attributes Automatically. When users do not provide any optimization hint to localize their cache attributes, our compiler can help to cache the proper attributes automatically for single relation access cases. We achieve this by analyzing users' code using domain-specific knowledge. In general, we will first try to satisfy the lowest occupancy constraint in GPU to determine the maximum size of shared memory per block. We then localize mesh attributes with different priorities. In the GPU backend, we prioritize the attributes involved in store operations first because they raise potential race conditions. We order the other attributes by their load frequency – it is more likely to cache the more frequently accessed attributes. In the CPU backend, we only cache the atomic-operation-related attributes. Detailed implementations are discussed in Section 6.2.

5.3 Memory Ordering Management

While mesh attribute localization brings a significant performance improvement by exploiting data locality using shared memory, moving the attributes between global memory and shared memory still has a non-negligible overhead. In the *prologue* and *epilogue* passes for attribute localization, the mapping from the local index space to the global index space is *out-of-order* as shown in Figure 6 (a), making the data ordering inconsistent between the shared memory and the global memory. We want a coalesced global memory access so that each access brings a big chunk of data to the shared memory at once.

Our compiler can reorder the global memory layout, aligning it with the local index space. To be more specific, we sort the global data attributing using the indices of the patches as the primary key and using the local indices within each patch as the secondary key. The ribbon elements are ignored during this sorting. By storing the mesh attributes in the global memory with the reordered layout, we can maximize the memory throughput as shown in Figure 6.

However, the reordered memory layout is not a silver bullet for all mesh attributes. First, each mesh attribute may be visited from different `mesh-for` loops that prefer different orders. Second, the

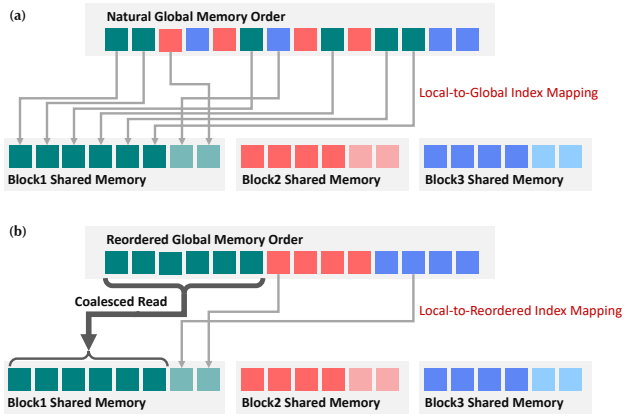


Fig. 6. Fetching data attributes from **(a)** the naturally-ordered global memory, or from **(b)** the reordered global memory. The reordered layout allows us to bring a bigger chunk of data into shared memory at once. The lighter-colored blocks in the shared memory represent the ribbon elements.

mesh attributes may need to interact with external non-mesh data ordered naturally. We, therefore, provide global memory reordering only as an option in the mesh definition phase. Users can experiment with different reordering schemes without changing the computation. For example, if we want to reorder the `bunny.verts.vel` and the `bunny.verts.force` while keep the order `bunny.verts.pos` as is, we only need to modify the line 3-5 in the mesh data definition from Section 4.1 as follows:

```
1 mesh.verts.place({'pos' : ti.math.vec3}, reorder=False)
2 mesh.verts.place({'vel' : ti.math.vec3,
3                   'force' : ti.math.vec3}, reorder=True)
```

We can also reorder the mesh attributes defined on other types of elements:

```
1 mesh.cells.place({'B' : ti.math.mat3,
2                   'w' : ti.f32}, reorder=True)
```

The default memory ordering for each attribute is the natural order. Note that users do not need to worry about the reordered indices even they want to perform index-based queries like mentioned in Section 4.3. Our compiler manages the reordering mappings implicitly. The memory ordering management is fully *decoupled* with computation code. Together with the `mesh_local` decorator, users can use these optimization hints to swiftly exploit the different memory orderings and different cached attributes at the same time. We refer to Section 7.3 to see the influence of memory orderings.

6 IMPLEMENTATION

This section presents the implementation of our compilation system, which abstracts the mesh-based operations and transforms the mid-level intermediate representations into the optimized executable low-level instructions for different architectures.

Our system is implemented upon Taichi [Hu et al. 2019] and its hierarchical static-single assignment (SSA) intermediate representation (IR) system. We extend the IR system for mesh-based operations.

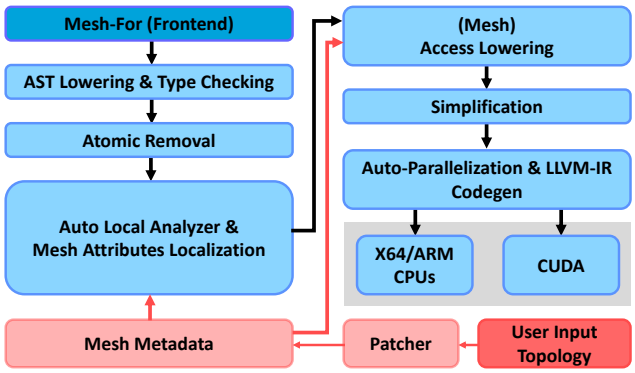


Fig. 7. The pipeline of our mesh compiler. The black arrows represent our mesh domain-specific IR transformation flow, and the red arrows indicate the use of patch-based mesh metadata.

The key components of our compiler include a localization transformation to optimize the attribute access, a mesh access lowering transform, and a parallel code generator. We also extend the atomic demotion and a simplification transform to remove the redundant accesses to mesh attributes and relations. Our system partitions the input mesh to generate the local relations and the index mappings for each patch. We call them the mesh metadata, which would be managed by the compiler implicitly. The compilation pipeline is summarized in Figure 7.

In the end, our code generation emits the LLVM IR [Lattner and Adve 2004] by just-in-time compilation. The IRs from our compiler would be compiled along with other original Taichi IRs to multiple backends such as x64, ARM64, and CUDA.

6.1 Domain-specific Intermediate Representations

The native Taichi IR statements only support grid-based operations, and they are not suitable to perform domain-specific optimizations on meshes. In this section, we introduce two important new IR statements for mesh-based operations.

Relation Access. The relation access should be highly abstracted to perform compiler-level analysis and transformations. We introduce the `RelationAccessStmt` statement to *unify* the representation for accessing both *static relations* and *dynamic relations*. Our language exposes a reference-style relation access interface in the frontend (e.g., `for v in e.verts`). Our compiler keeps track of the indices. Since the input mesh are partitioned into patches, we only need the *local indices* to visit the local relations. To be more specific, our `RelationAccessStmt` takes its *from-end* element's *local index* as input and return the *local index* of its *to-end* element.

Index Space Conversion. The `RelationAccessStmt` statement works only on the local index space for all relations inside a patch. We, therefore, introduce the `IndexConversionStmt` statement to convert the local indices to global ones. Note that we allow the users to choose a reordered global memory layout. Our `IndexConversionStmt` will also convert the local indices to the reordered global indices if necessary. We also reuse the `GlobalPrtStmt` in Taichi to visit the

global memory. The `GlobalPrtStmt` will return us the global memory address of a mesh attribute once it gets the global index of that attribute using `IndexConversionStmt`.

6.2 Localization Transformation

Our compiler localizes the mesh data using our domain-specific IRs. We separate the localization process into two stages: an analysis stage and a transformation stage.

The analysis stage decides which attributes shall be cached. We will try to cache the hinted attributes whenever is possible. In this case, we can loop for the `mesh_local` decorator to locate those attributes. If the compiler can not find any user-specified attributes in a `mesh-for` loop, it will collect all the attributes involved in the `IndexConversionStmt` statements. We evaluate the priorities of these attributes based on their access types and frequencies. For example, we will prioritize the attributes involved in atomic writes because the atomic operations in global memory are expensive. Due to the trade-off between the shared memory size and the occupancy, we set the lowest occupancy which activates at least 2 blocks per streaming multiprocessor as the constraint. We use this constraint to estimate the maximum shared memory size per block and cache the mesh attributes according to their priorities.

The transformation stage is used to turn the global memory accesses into local ones. In this stage, the wanted attributes will first be grouped together based on their element type. The compiler will then generate the *prologue* and *epilogue* passes for each mesh element type and its related attributes. The *prologue* pass loads the attributes into shared memory before the computations, whereas the *epilogue* pass writes the computed attributes back to global memory. To ensure data consistency, we use `__syncthreads` after *prologue* and before *epilogue* for the GPU backend.

The localization process is illustrated as the "Auto Local Analyzer & Mesh Attributes Localization" block in Figure 7. Since all the wanted attributes are in shared memory after the localization process. We safely drop the `IndexConversionStmt` statements and replace each `GlobalPtrStmt` statement to another localized statement `BlockLocalPtrStmt` in Taichi. The `BlockLocalPtrStmt` statement returns the address of the shared memory for the GPU backend. Instead of explicitly controlling caches as we do on GPUs, on CPUs we allocate a memory buffer small enough to fit in L1 data cache, and let the hardware control data residency inside the buffer. The `BlockLocalPtrStmt` statement will return the address of the small memory buffer to perform frequent reads and writes. Since we frequently read/write to data in the buffer, they will reside in L1 data cache with high probability, serving a similar role as data in GPU shared memory.

6.3 Access Lowering

In this access lowering process, all the mesh-related statements such as `RelationAccessStmt` and `IndexConversionStmt` statements are transformed into lower-level statements that compute the memory address offset and visit the physical memory, guided by the mesh metadata. The lowered memory access may produce redundant low-level statements, and we rely on another simplification process to eliminate them.

6.4 Mesh Metadata

The compiler workflow aforementioned depends on the mesh metadata as illustrated using the bold red arrows in Figure 7. The metadata, including the index mappings, local relations, and element numbers, are generated for each patch. The index mappings bookkeep the index transformation. The local relations store the indices of the *from-end* and *to-end* elements of relations in the *local index* space. For local *static relations*, we only need to pack the values together because each *from-end* element has a static number of neighbors of *to-end* neighbors. For local *dynamic relations*, we use two packed arrays for both the values and the offset. We also keep the number of owned elements and total elements for each element type in the metadata. We primarily use the number of owned elements for each patch to decide the range of a `mesh-for` loop. We also use these sizes to locate other metadata because we use a packed mode to store them patch-by-patch. The packed storage scheme of the metadata aligns with the order of `mesh-for` loops. This further helps reduce the fetching overhead for these metadata.

7 DESIGN VALIDATIONS

In this section, we validate our compiler design decisions by testing different options. We choose spring-force and vertex normal computations as two simple micro-benchmarks in this section. Because these experiments have relatively simple computation kernels, the performance of these experiments can better reflect the memory access overhead under different settings.

7.1 Choice of Mesh Partitioning Strategies

Mesh partitioning is an important pre-computation step for our compiler. In this section, we validate our mesh partitioning algorithm and discuss the influence of patch size.

We set up two mass-spring simulation experiments with a tetrahedron mesh consisting of 226, 857 vertices and 1, 231, 193 edges, and a triangle mesh consisting of 172, 974 vertices and 518, 916 edges. We measure the spring force evaluation time on those two meshes. For each experiment, we use two different styles to compute spring force: one scattering style implementation involving a *static relation* query ($\mathcal{EV}$ in this case), and one gather style implementation involving a *dynamic relation* query ($\mathcal{VV}$ in this case). We test two different patching algorithms: k-means and our greedy-based method with various of patch size options.

As we can see from Figure 8, our greedy algorithm produces comparable results for triangle mesh, and improves the performance by approximately 20% for tetrahedron mesh, compared with k-means. The performance curve aligns with the normalized theoretical global memory access γ for each mesh. That confirms our hypothesis in Section 5.1 that fewer ribbon elements leads to higher performance. Our compiler generally prefers larger patch sizes as larger patches tend to have fewer ribbon elements. But larger patches also reduces the number of resident blocks per stream-multiprocessor. We pick 2048 as the default patch size for our compiler (and also used this default size to produce all other experiments in this paper). Note that we do not need to compute any relation during run time, the increase of patch size will affect only occupancy but not the relation acquisition time.

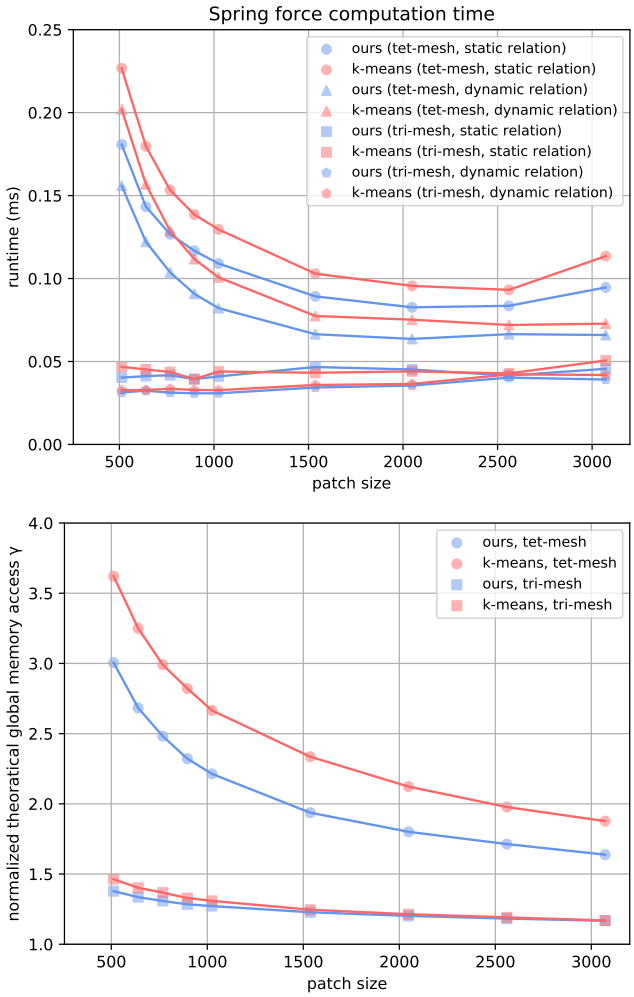


Fig. 8. The performance (**top**) and the normalized theoretical global memory access γ (**bottom**) with respect to different patching strategies and different patch sizes.

7.2 Choice of Localized Data

The key idea to improve the performance for mesh-based operations is to improve data locality by moving data to on-chip memory before they are needed for computations. Our system localizes only the mesh attributes but not the relations as described in Section 5.2. We validate this strategy using the volumetric mass-spring system simulation described in Section 7.1.

We test four different localization strategies: (1) leaving everything to hardware scheduling, (2) caching relation data only, (3) caching attribute data only, and (4) caching both relation and attribute data. We run each experiment for 25, 000 times and report the average timing spent on the force computation kernel. The results are shown in Figure 9.

Caching nothing strategy has the slowest performance. This meets our expectation that hardware scheduling is not enough for mesh-based operations. The unstructured memory access leads to frequent

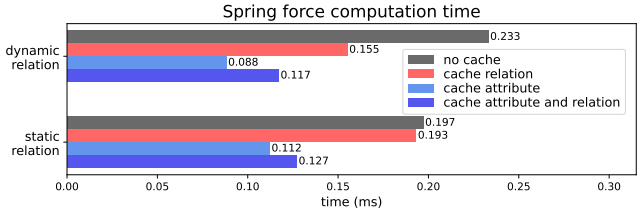


Fig. 9. Timings for different caching strategies. We tried cache nothing and leave everything to hardware scheduling, cache relation only, cache attribute(s) only and cache both relation and attribute.

cache misses. And the data pre-fetched by hardware is not enough to revolve these cache misses. This behavior calls for a specifically tailored solution for meshes.

Caching relations strategy helps improving the performance. To be more specific, it helps the performance more for *dynamic relations*. We prepared the pre-computed relations in compile-time, every relation is loaded only once in this example and most relation loading patterns align with their memory ordering. This coherent relation loading applies to most mesh-based applications as well. The static relation visits knows the addresses and offsets because every element has known *to-end* neighbors in these relations. (e.g. an edge $\mathcal{E}$ must have two vertices neighbors $\mathcal{V}$.) Different threads inside a patch can visit global memory in a coalesced batch even if relations are not cached by the compiler. That is why the improvement from caching relations for static relations is marginal. On the contrary, different threads are not able to load dynamic relations from global memory in coalesced visits because the number of *to-end* elements are not fixed. Our implementation of this caching relation strategy pre-fetchs relations from global memory to shared memory in batches regardless of the relation types. So caching dynamic relations has a more significant performance gain. It is worth mentioning that this caching relation strategy is the closest strategy to RXMesh. The only difference is that we pre-compute relations in compile-time. RXMesh, on the contrary, performs relation computations on the fly, so it needs to access multiple relation tables in run-time. Hence caching relations becomes an important design for RXMesh, but it is not the best strategy for our system.

Caching attributes strategy is the most efficient one in our test cases. This is based on our observation that mesh attributes are often accessed in unstructured ways, and each mesh attribute is very likely to be accessed by multiple local computations. For example, the position attribute of a vertex in a mass-spring system may be visited by all of its vertex/edge neighbors to compute spring forces. Caching these attributes not only makes their load and store operations local, but also reduces data races among patches. When writing back attributes to global memory after all local computations, only those attributes of ribbon elements may encounter write conflicts. Therefore, caching attributes can provide us a significant amount of performance gain.

Caching both attributes and relations strategy seems to be a perfect solution in our first thought. However, the size of shared memory is limited so larger shared memory consumption will lead to fewer resident blocks per stream-multiprocessor, lowering occupancy in

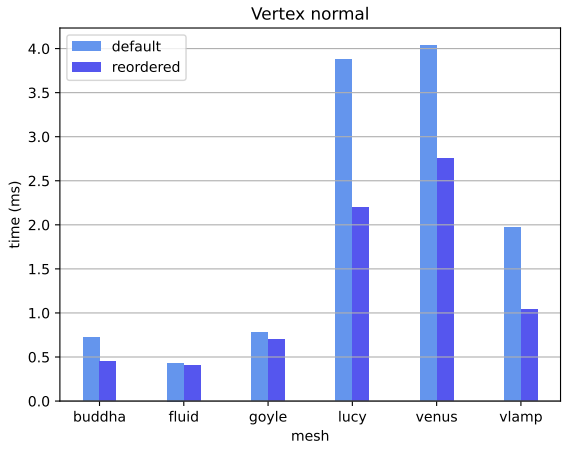


Fig. 10. Timings for different memory orderings. We use the same settings as in Section 8.2 to compute the vertex normals. We set `reorder=True` for the reordered cases.

GPU. As shown in the last bar in Figure 9, this strategy is less competent compared to caching attributes only.

These experiments indicate that on the one hand, caching proper data into shared memory will improve the performance for mesh-based operations, while on the other hand, abusing shared memory with everything we can cache can hurt the performance too. We observe that the size of relations and attributes are usually with the same order of magnitude in most of our applications. Therefore, we decide to only cache data attributes but not relations. We also allow users to try different cached attributes by playing with the optimization hint `mesh_local`, in case they want to fine-tune a perfect set of cached attributes.

7.3 Influence of Memory Orderings

We cache mesh attributes to accelerate the data access for computations in mesh-based operations. However, our compiler still need to move the attributes between global and shared memory in *prologue* and *epilogue* stages. A good memory ordering will help to accelerate our data exchange to coalesced read and writes. Here we show our performance under different memory orderings using a vertex normal application. The tested surface meshes are shown in Figure 14 and Figure 15. To change memory ordering from natural order to a patch-aligned order decided by our compiler, we only need to set `reorder=True` when defining the mesh data type as described in Section 5.3. Figure 10 shows that our system takes advantage of coalesced data access from the global memory. Note that the speedup varies among different meshes, one might want to experiment with different memory ordering options. Our compiler allows users to explore different memory orderings for different attributes as described in Section 5.3 without changing the computation code.

8 RESULTS

In this section, we demonstrate several applications implemented in our system, comparing our system with other mesh-based libraries, and high-performance implementations using other languages.

We collect the triangle meshes from Thingi10K [Zhou and Jacobson 2016] and use TetGen [Si 2015] to generate the corresponding tetrahedron meshes. We also use OpenVDB [Museth et al. 2013] to generate some more structured triangle meshes. We use the remesh geometry node in SideFX Houdini to produce meshes with different resolutions.

The mesh-grid hybrid simulation shown in Figure 1 runs on a single NVIDIA A100 Tensor Core GPU with 80GB of device memory. All other GPU experiments are tested on Nvidia RTX 3090 with 24GB device memory. The GPU timings are measured using the NVIDIA profiling tool (`nvprof`), which keeps track of the kernel time. The CPU experiments are executed on an 11th Gen Intel(R) Core(TM) i7-11700F with 8 cores at 2.50GHz and 64GB of main memory. Detailed run time and compile time for all all examples in this section can be seen in Table 1 and Table 2 in Appendix A.

To make the comparisons fair, we disable memory reordering for all experiments in this section. We also fine-tuned the hyper-parameters such as block-dim for both our system and all other baseline methods independently in every experiment to ensure fairness.

8.1 Mass-Spring System

We implemented a mass-spring system with both explicit and semi-implicit [Baraff and Witkin 1998] time integration schemes. For the explicit integration, the key step is to compute the force for each vertex. We tested two versions to perform the force evaluation. The first version is based on a *scattering-style* operation using the $\mathcal{EV}$ relation where we loop over the edge elements of the simulated model, compute the spring force for each edge and scatter the force to the endpoints. This is an intuitive implementation because we only need to compute the spring force once for each spring. The applied spring forces to its endpoints are equal in magnitude and opposite in direction as described in Newton's third law of motion. Another way to evaluate forces is based on a *gathering-style* operation using the $\mathcal{VV}$ relation where we loop over all the vertex elements and compute the force for each vertex based on the positions of all its neighbor vertices. Although this method would require two force evaluations for every spring, it avoids write-conflicts hence can be more efficient in many cases. We implemented the implicit integration scheme using 10 matrix-free conjugate gradient iterations per frame. By matrix-free, we mean that we do not store the Hessian matrix anywhere, but evaluate the Hessian blocks on the fly during the matrix-vector multiplication inside the CG iterations instead. We used *gathering-style* operations to perform both the force evaluation and the matrix-vector multiplication for the implicit integration. We generated varying sized bunny models (from 1×10^4 to 5×10^6 vertices) and ran all the experiments using our system and other programming languages or data structures on GPU as shown in Figure 11. Note that Simit [Kjolstad et al. 2016] does not care about whether an operation scatters or gathers the data attributes, because it assembles its matrix directly from the graph representation of

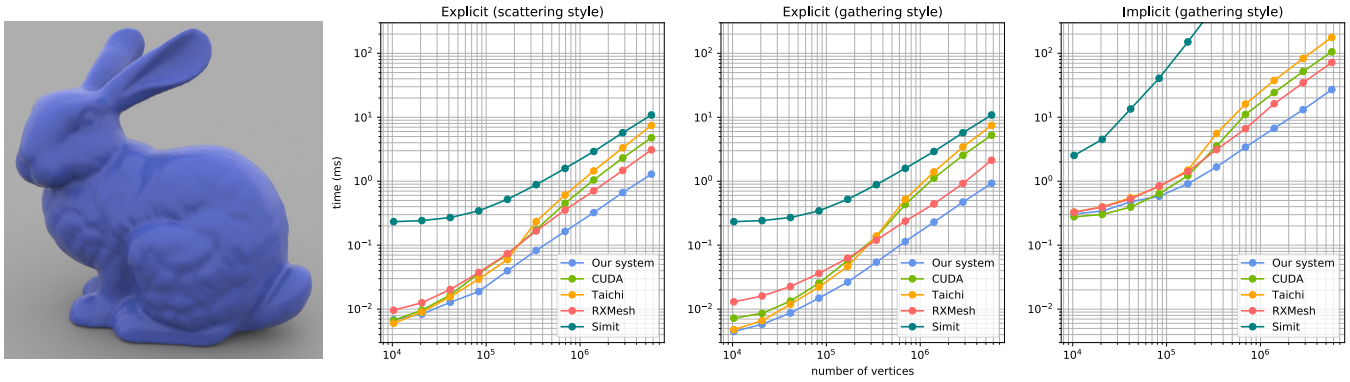


Fig. 11. The GPU timings of simulating a surface-only bunny model with different resolutions and integration schemes collapsing onto the ground. **Left:** The bunny model. **Right:** The GPU timings of our system, CUDA, Taichi, RXMesh, and Simit on different cases. We apply a *scattering-style* force computation scheme using the $\mathcal{EV}$ relation and a *gathering-style* force computation scheme using the $\mathcal{VV}$ relation to implement the explicit time integration. We use the *gathering-style* scheme to implement the implicit integration where we applied 10 matrix-free conjugate gradient iterations to solve the linear system.

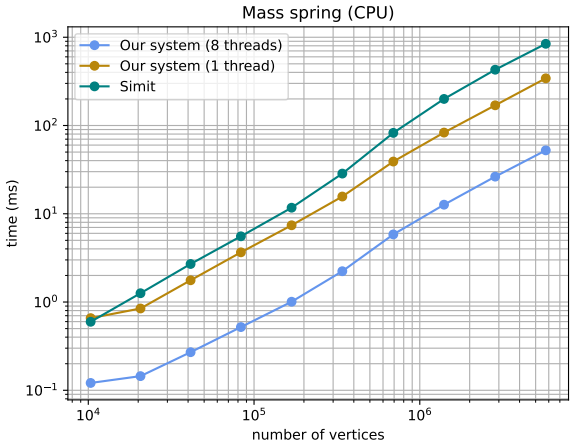


Fig. 12. The CPU timings of our system with eight threads, our system with one thread, and Simit for the same bunny example shown in Figure 11. The force is computed with gathering-style operations in our system.

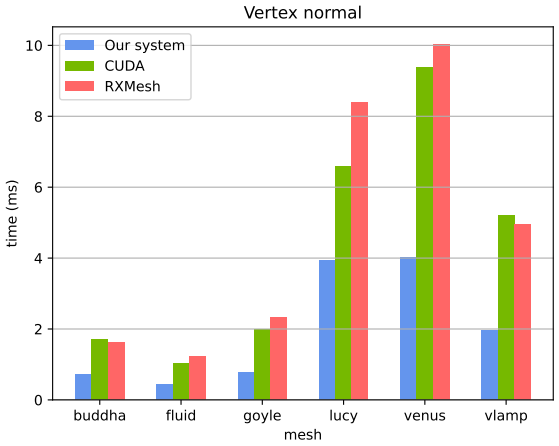


Fig. 13. The GPU performance of the vertex normal examples. We compare our system with an optimized hardwired CUDA implementation provided by the authors of RXMesh and an RXMesh implementation.

a mesh. Therefore the performance for Simit is the same for both scattering and gathering styles in the explicit integration. Our system achieves the best performance in all the experiments. For high resolution mesh models, our system is more than twice faster (2.3× for explicit simulations and 2.7× for implicit simulations) compared to the most efficient competitor on GPU (RXMesh).

We also ran our mass-spring simulations on CPU, comparing the CPU performance of our system and Simit which is the state-of-the-art mesh compiler on CPU. The results are shown in Figure 12. Our system shows its advantage as the size of the mesh increases. Our compiler performs an order of magnitude faster than Simit for all tested meshes. Because Simit runs with only one thread, we tested our system on one thread as well. The gap between the dark green curve and dark golden curve in Figure 12 indicates that the CPU performance of our system comes not only from multi-threading, but from the well-exploited data locality too. In high-resolution cases, the single-threaded and eight-threaded versions of our system are 2.46× and 16.11× faster than Simit respectively.

8.2 Vertex Normal

Vertex normal computation is a simple but typical mesh-based operation. It is widely used to render deformable objects. This operation first loops over all the surface triangle elements and *gathers* the positions from its vertices, then computes the surface normal using a cross-product and *scatters* the normal to the vertices using the $\mathcal{FV}$ relation. We implemented the weighted vertex normal [Max 1999] algorithm using our language. We pick several high-resolution surface models for this experiment which are compiled in Figure 14

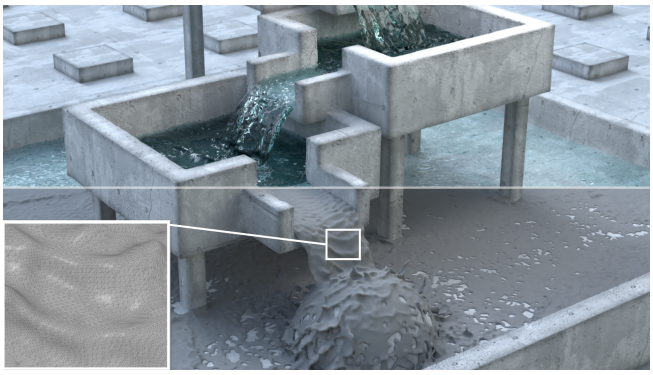
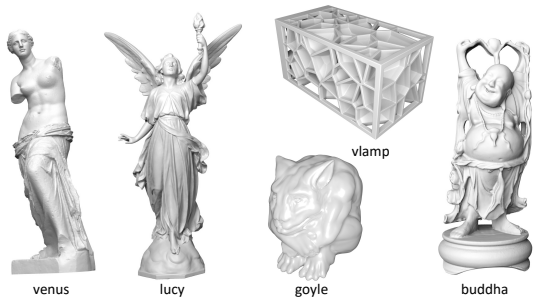


Fig. 14. An OpenVDB-reconstructed mesh we used in the vertex normal example.



model	#verts.	#edges	#faces
buddha	4, 059, 506	12, 179, 136	8, 119, 424
goyle	6, 291, 458	18, 874, 368	12, 582, 912
lucy	20, 237, 202	60, 711, 600	40, 474, 400
venus	25, 232, 882	75, 698, 640	50, 465, 760
vlamp	9, 559, 520	28, 678, 554	19, 119, 232

Fig. 15. The surface models we used in the vertex normal and geodesic distance applications.

and Figure 15. We compared our system against RXMesh [Mahmoud et al. 2021], as well as an optimized hardwired CUDA implementation also provided by the authors from RXMesh. In this example, both our system and the CUDA implementation use pre-computed $\mathcal{FV}$ relations and store them in global memory. RXMesh saves $\mathcal{FE}$ and $\mathcal{EV}$ relations in shared memory and computes the wanted $\mathcal{FV}$ on the fly. As shown in Figure 13, our system outperforms both the optimized CUDA implementation and RXMesh.

8.3 Geodesic Distance

The geodesic distance application computes the shortest distance between a selected vertex and all other vertices on a surface. We implemented the minimalistic parallel algorithm [Calla et al. 2019] to approximate the geodesic distance. The algorithm comes with two passes. In the first pass, it generates a topological level set around the selected vertex. In the second pass, it iteratively activates a small set of vertices according to the level set and updates the geodesic distance and error of these vertices in parallel. The second pass keeps activating new vertices and deactivating old vertices with

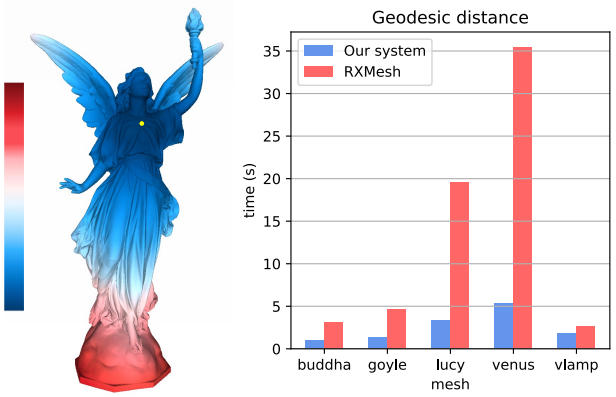


Fig. 16. Geodesic distance. **Left:** The normalized geodesic distance to a selected vertex colored in yellow, visualized from blue (short) to red (long). **Right:** The performance of our system and RXMesh with different input meshes.

low errors. We compared our system with RXMesh using the same meshes used in the vertex normal application as shown in Figure 15. Our system has a significant speedup compared with RXMesh, as shown in Figure 16.

8.4 Projective Dynamics

Projective dynamics [Bouaziz et al. 2014] is an efficient simulation framework for elastic materials. We wrote an elastic body simulation as shown in Figure 17. Instead of using a pre-factorized system matrix in the vanilla implementation of projective dynamics on CPU, we decided to pre-compute the system matrix and perform 5 local-global iterations per frame and 30 conjugate gradient (CG) iterations per global step, taking full advantage of the parallelism on GPU. The bottleneck of this simulation is mostly on the matrix-vector multiplications inside the CG iterations. Since our compiler provides a flexible interface to let users store data attributes on any types of mesh elements, we place the sparse system matrix of projective dynamics on both vertices and edges. The diagonal terms of the system matrix are stored on vertices and off-diagonal terms are stored on edges. Our system achieves a 34 fps simulation in real-time as shown in Figure 17.

Our conjugate gradient implementation involves 1 matrix-vector multiplication, 2 vector dot-products and 3 vector additions every iteration. Synchronization time is also non-negligible especially for matrix-vector multiplication and dot-products. The pie chart in the left of Figure 17 reports the cost breakdown to simulate one frame of projective dynamics using our system where mesh-based operations (including force computation and part of matrix-vector multiplication) takes 42.73% of the run time. We also implemented a Taichi version of the same projective dynamics algorithm as a baseline solution. Our system achieves 1.61× speedup compared to Taichi for these mesh-based operations and 1.26× speedup for overall simulation. In this projective dynamics example, our system accelerates the simulation from 27 fps to 34 fps to enable real-time interactivity at the cost of 18.4 seconds in compiling.

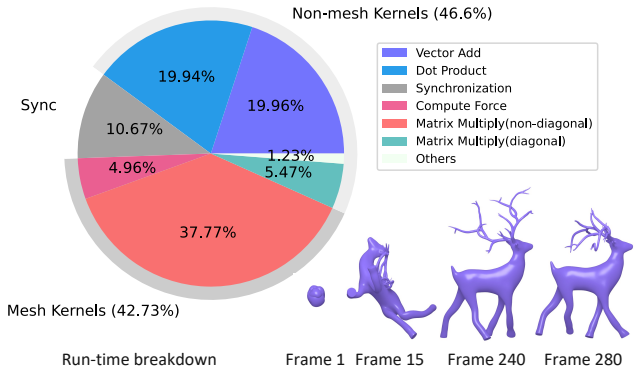


Fig. 17. An elastic body simulation of a deer model with 194, 267 vertices, 1, 047, 693 edges and 701, 496 cells based on projective dynamics (PD). The simulation runs 5 PD iterations per frame and 30 CG iteration per global step at 34 fps in real-time. **Left:** Run-time cost breakdown to simulate one frame of PD using our system. **Right:** The PD solver recovers the deer model from a randomly initialized state to its rest pose quickly.

8.5 XPBD Cloth Simulation

We implemented an array of cloth simulation using eXtended Position Based Dynamics (XPBD) [Macklin et al. 2016]. We assign 10, 000 random vertices for each cloth independently and tessellate it using Delaunay triangulation. The simulation runs on 100 pieces of cloth consists of 1, 000, 000 vertices, 2, 997, 248 edges and 1, 997, 348 faces in total. We use spring lengths to set up stretch constraints and use dihedral angles to set up bending constraints as described in PBD [Müller et al. 2007]. Each cloth is assigned with unique stretch and bending compliance as shown in the top of Figure 18.

As a result, we achieve 0.725s per frame to run 167 substeps where each substep consists of 5 nonlinear Jacobi iterations. We iterate through all the stretch and bending constraints in every Jacobi iteration. Each stretch constraint is updated with an $\mathcal{EV}$ relation and each bending constraint is updated with an $\mathcal{EF}$ relation to compute the dihedral angle and an $\mathcal{FV}$ relation to scatter the vertex displacement. The stretch and bending constraints update takes 35.6% and 53.5% of the simulation time respectively.

We compare our system with a Taichi implementation. Our system achieves 1.27× speedup for stretch constraints, 1.60× speedup for bending constraints, and 1.42× speedup for overall simulation compared to Taichi. We also compare our system with an RXMesh implementation. Since RXMesh does not support multiple relation access within one kernel, we are not able to compare the performance of bending constraints. Our system achieves 3.75× speedup for stretch constraints compared to RXMesh as shown in the bottom of Figure 18. In this example, the cloth elements are implicitly grouped by the cloth ID, improving the default data locality. This dilutes the advantages gained from patch-based representations including our system and RXMesh. Nevertheless, our system still performs better compared with both Taichi and RXMesh.

8.6 Mesh-Grid Hybrid Simulation

To exploit the scalability and flexibility of our system, we implemented soft body simulation using material point method with corotated linear elastic materials. The force model is evaluated from a Lagrangian point of view using meshes [Jiang et al. 2015]. We set 17, 010 armadillos in the scene with 222, 048, 540 vertices and 713, 739, 600 cells in total. In each time step, we first compute the hyper-elastic force for each vertex using mesh information, then interpolate the momentum and mass onto the grid. Once the velocities on the grid are computed, we interpolate them back to the mesh vertices using MLS-MPM [Hu et al. 2018a]. We used a Taichi implementation of this simulation as our baseline reference. Taichi takes 2.9 minutes on average to produce 300 substeps for every frame. Within each frame, the force computation step takes 50 seconds which is 28.7% of total simulation time. Our system achieves 2.58× speedup for this mesh-based operation, reducing its time to 19 seconds per frame. As a result, our system achieves 1.21× speedup for overall simulation compared to Taichi and takes 2.4 minutes on average to produce 300 substeps for this mesh-grid hybrid simulation. Our system takes 3.5 minutes to compile and saves 2.5 hours at run time to generate a 10-second video for this high-resolution simulation.

8.7 Performance for Different Mesh Operations

We tested the performance of our method using 32 different mesh-based operations which is the composition of 16 different types of relations and 2 attribute accessing styles as shown in Figure 19. In all experiments, we define a data attribute sized 40 bytes for all elements in a mesh including vertices, edges, faces, and cells. For the gathering-style access, we loop over the *from-end* elements of a relation, and accumulate their *to-end* neighbors' 40-byte attribute to themselves. For the scattering-style access, we loop over the *from-end* elements, and scatter their 40-byte attributes to their *to-end* neighbors.

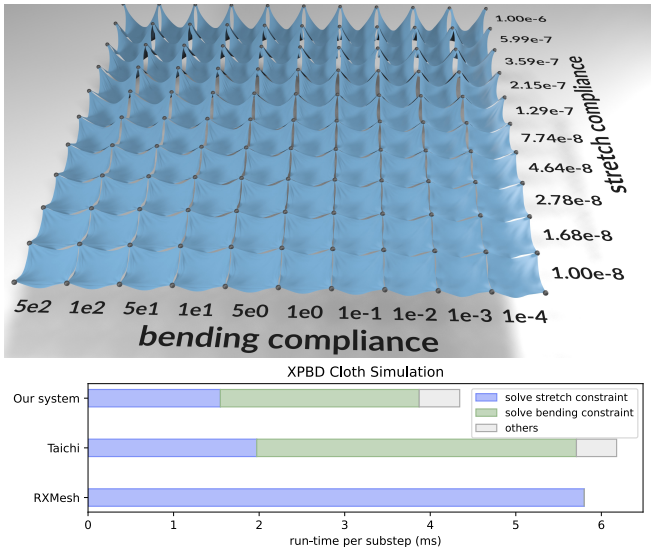
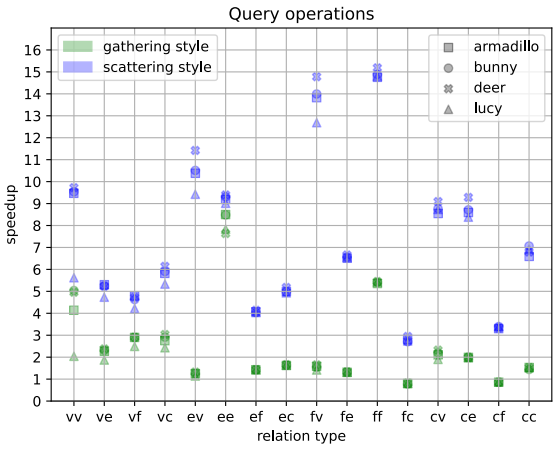


Fig. 18. **Top:** Simulating 100 pieces of cloth with different consisting of 1, 000, 000 vertices, 2, 997, 248 edges and 1, 997, 348 faces using XPBD. **Bottom:** Simulation runtime for each substep using our system, Taichi and RXMesh.



model	#verts.	#edges	#faces	#cells
armadillo	1, 753, 678	9, 198, 712	13, 482, 531	6, 037, 496
bunny	1, 783, 121	9, 444, 504	13, 916, 746	6, 255, 362
deer	2, 089, 717	11, 223, 724	16, 645, 666	7, 511, 658
lucy	1, 065, 930	5, 770, 116	8, 588, 004	3, 883, 817

Fig. 19. Speedup of our method over a baseline Taichi implementation for all query operations on different input meshes. The gathering- and scattering-style data accesses are colored in green and blue respectively. The models are listed in the bottom.

We wrote a Taichi implementation as the baseline because we built our compiler upon Taichi. Any performance gain or loss in Figure 19 comes from our mesh-oriented modifications. These micro-benchmarks validate that our improvements upon Taichi brings a considerable performance gain.

The experiments also show that our compiler generally excels at most mesh-based operations regardless of the queried relations or the attribute accessing styles. A precise observation we made from Figure 19 is that our system accelerates the scattering-style operations more compared with the gathering-style ones. That is because these scattering-style operations are very likely to cause data races. Our compiler stores the mesh attributes back to global memory only during the *epilogue* pass so that most inter-patch data conflicts are avoided.

9 LIMITATIONS AND FUTURE WORK

Currently, the topology of input meshes needs to be static, so that we can generate the compiler metadata before run time. Supporting meshes with dynamically modified topology is an exciting research problem. Also, our test cases may not cover all common mesh types. We would like to test more types of meshes in the future.

We focus on local mesh operations for now, where we assume all computations happen on the one-ring neighborhood of a mesh element. Users need to perform the `mesh-for` operations multiple times to pass the information beyond one-ring neighbors. Collisions and contacts involve lots of run-time information, hence they are hard to optimize using our compiler. We use an external array to store dynamically generated relations due to collisions, and we visit the contact information using a regular range-for instead of our mesh-for loops. Our current system accelerates only the static-mesh-related computation and does not affect dynamic relations generated by contacts. We plan to investigate the possibility of supporting higher-order relations and dynamically generated relations in the future.

The patching algorithm is another promising direction to improve. We observe that our compiler prefers lower ribbon ratios to achieve better performance. But our patching algorithm remains a raw approximation to achieve this goal. We plan to investigate some other graph partitioning algorithms such as sparse cut to generate better patches. Our current patching algorithm is implemented on CPU. This causes fairly long compile time if the compiler cannot be warm-started using cached metadata as reported in Appendix A. We plan to accelerate the mesh partitioning process with a GPU partitioning implementation.

We build our system upon the Taichi programming language and it would be a promising line of future work to integrate Taichi's automatic differentiation [Hu et al. 2020] into our mesh compiler via source code transformation. We believe this potential extension would benefit many deep learning tasks on meshes.

Our compiler supports multi-core CPU and GPU backends for now. Extending the supported backends to other parallel architectures such as multi-GPU or MPI would be an interesting line of improvement as well. We are also interested in exploring other GPU-based code generations, for example, SPIR-V and OpenCL.

10 CONCLUSIONS

We present MeshTaichi, a new compiler to make mesh-based operations efficient. We achieve high performance by exploiting data locality for mesh attributes. To be more specific, we partition meshes into patches and transform wanted attributes for each patch to on-chip memory so that the mesh-based operations can access their data faster. Our compiler provides an intuitive programming interface where users can program concise and intuitive code without worrying about mesh relations and attribute indices. Our compiler also decouples low-level optimization options from computation, enabling users to exploit different data orderings and cached attributes without changing the computation. Our compiler supports various mesh operations for both triangle and volumetric meshes and generates code for both CPU and GPU backends. As a result, our CPU solution is an order of magnitude faster compared to the best mesh compiler on CPU Simit; our GPU solution is 1.4 to 6 times faster than the state-of-the-art mesh data structure RXMesh.

ACKNOWLEDGMENTS

We thank Mingrui Zhang and Chuqiao Zhou for early-stage brainstorming, Xiuqi Yang for performance profiling, Jihua Liu and Shumu Xu for providing the OpenVDB-reconstructed mesh, Yun (Raymond) Fei and Ming Gao for discussions on atomic operations, Ahmed H. Mahmoud for answering our questions about RXMesh, and Haidong Lan and Bo Qiao for proofreading. We also thank the anonymous reviewers for their constructive feedback.

REFERENCES

David Baraff and Andrew Witkin. 1998. Large steps in cloth simulation. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*. 43–54.

Bruce G Baumgart. 1972. *Winged edge polyhedron representation.* Technical Report. STANFORD UNIV CA DEPT OF COMPUTER SCIENCE.

Gilbert Louis Bernstein and Fredrik Kjolstad. 2016. Perspectives: Why New Programming Languages for Simulation? *ACM Transactions on Graphics (TOG)* 35, 2 (2016), 1–3.

Gilbert Louis Bernstein, Chinmayee Shah, Crystal Lemire, Zachary Devito, Matthew Fisher, Philip Levis, and Pat Hanrahan. 2016. Ebb: A DSL for physical simulation on CPUs and GPUs. *ACM Trans. Graph.* 35, 2 (2016), 21:1–21:12.

Volker Blanz and Thomas Vetter. 1999. A morphable model for the synthesis of 3D faces. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*. 187–194.

Mario Botsch, Stephan Steinberg, Stephan Bischoff, and Leif Kobbelt. 2002. Openmesh-a generic and efficient polygon mesh data structure. (2002).

Sofien Bouaziz, Sebastian Martin, Tiantian Liu, Ladislav Kavan, and Mark Pauly. 2014. Projective dynamics: Fusing constraint projections for fast simulation. *ACM transactions on graphics (TOG)* 33, 4 (2014), 1–11.

Ajay Brahmakshatriya, Emily Furst, Victor A Ying, Claire Hsu, Changwan Hong, Max Ruttenberg, Yunming Zhang, Dai Cheol Jung, Dustin Richmond, Michael B Taylor, et al. 2021. Taming the Zoo: The unified GraphIt compiler framework for novel architectures. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 429–442.

Luciano A Romero Calla, Lizeth J Fuentes Perez, and Anselmo A Montenegro. 2019. A minimalistic approach for fast computation of geodesic distances on triangular meshes. *Computers & Graphics* 84 (2019), 77–92.

Swen Campagna, Leif Kobbelt, and Hans-Peter Seidel. 1998. Directed edges—a scalable representation for triangle meshes. *Journal of Graphics tools* 3, 4 (1998), 1–11.

He Chen, Hyojoon Park, Kutay Macit, and Ladislav Kavan. 2021. Capturing Detailed Deformations of Moving Human Bodies. *arXiv preprint arXiv:2102.07343* (2021).

Mathieu Desbrun, Mark Meyer, Peter Schröder, and Alan H Barr. 1999. Implicit fairing of irregular meshes using diffusion and curvature flow. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*. 317–324.

Zach DeVito, Niels Joubert, Francisco Palacios, Stephen Oakley, Montserrat Medina, Mike Barrientos, Erich Elsen, Frank Ham, Alex Aiken, Karthik Duraisamy, Eric Darve, Juan Alonso, and Pat Hanrahan. 2011. Liszt: A Domain Specific Language for Building Portable Mesh-based PDE Solvers. *Proceedings of 2011 SC - International Conference for High Performance Computing, Networking, Storage and Analysis* 3, 9.

Antonio DiCarlo, Alberto Paoluzzi, and Vadim Shapiro. 2014. Linear algebraic representation for topological structures. *Computer-Aided Design* 46 (2014), 269–274.

Yu Fang, Minchen Li, Ming Gao, and Chenfanfu Jiang. 2019. Silly rubber: an implicit material point method for simulating non-equilibrated viscoelastic and elastoplastic solids. *ACM Transactions on Graphics (TOG)* 38, 4 (2019), 1–13.

Ming Gao, Xinlei Wang, Kui Wu, Andre Pradhana-Tampubolon, Eftychios Sifakis, Yuksel Cem, and Chenfanfu Jiang. 2018. GPU Optimization of Material Point Methods. *ACM Trans. Graph. (Proc. SIGGRAPH Asia)* 32, 4 (2018), 102.

Leonidas Guibas and Jorge Stolfi. 1985. Primitives for the manipulation of general subdivisions and the computation of Voronoi. *ACM transactions on graphics (TOG)* 4, 2 (1985), 74–123.

Yuanming Hu, Luke Anderson, Tzu-Mao Li, Qi Sun, Nathan Carr, Jonathan Ragan-Kelley, and Frédo Durand. 2020. DiffTaichi: Differentiable Programming for Physical Simulation. *ICLR* (2020).

Yuanming Hu, Yu Fang, Ziheng Ge, Ziyin Qu, Yixin Zhu, Andre Pradhana, and Chenfanfu Jiang. 2018a. A moving least squares material point method with displacement discontinuity and two-way rigid body coupling. *ACM Trans. Graph. (Proc. SIGGRAPH Asia)* 37, 4 (2018), 150.

Yuanming Hu, Tzu-Mao Li, Luke Anderson, Jonathan Ragan-Kelley, and Frédo Durand. 2019. Taichi: a language for high-performance computation on spatially sparse data structures. *ACM Transactions on Graphics (TOG)* 38, 6 (2019), 201.

Yuanming Hu, Jiafeng Liu, Xuanda Yang, Mingkuan Xu, Ye Kuang, Weiwei Xu, Qiang Dai, William T. Freeman, and Frédo Durand. 2021. QuanTaichi: A Compiler for Quantized Simulations. *ACM Trans. Graph.* 40, 4, Article 182 (July 2021), 16 pages.

Yixin Hu, Qingnan Zhou, Xifeng Gao, Alec Jacobson, Denis Zorin, and Daniele Panozzo. 2018b. Tetrahedral meshing in the wild. *ACM Trans. Graph.* 37, 4 (2018), 60–1.

Markus Ihmsen, Nadir Akinci, Markus Becker, and Matthias Teschner. 2011. A parallel SPH implementation on multi-core CPUs. In *Computer Graphics Forum*, Vol. 30. Wiley Online Library, 99–112.

Chenfanfu Jiang, Theodore Gast, and Joseph Teran. 2017. Anisotropic elastoplasticity for cloth, knit and hair frictional contact. *ACM Transactions on Graphics (TOG)* 36, 4 (2017), 1–14.

Chenfanfu Jiang, Craig Schroeder, Andrew Selle, Joseph Teran, and Alexey Stomakhin. 2015. The affine particle-in-cell method. *ACM Transactions on Graphics (TOG)* 34, 4 (2015), 1–10.

Petr Kadleček, Alexandru-Eugen Ichim, Tiantian Liu, Jaroslav Křivánek, and Ladislav Kavan. 2016. Reconstructing personalized anatomical models for physics-based body animation. *ACM Transactions on Graphics (TOG)* 35, 6 (2016), 1–13.

Lutz Kettner and Fernando Cacciola. 2006. Halfedge data structures. *CGAL User and Reference Manual* 3 (2006).

Fredrik Kjolstad, Shoaib Kamil, Jonathan Ragan-Kelley, David I. W. Levin, Shinjiro Sueda, Desai Chen, Etienne Vouga, Danny M. Kaufman, Gurtej Kanwar, Wojciech Matusik, and Saman Amarasinghe. 2016. Simit: A language for physical simulation. *ACM Trans. Graph.* 35, 2 (2016), 20:1–20:21.

Fredrik Berg Kjolstad and Marc Snir. 2010. Ghost cell pattern. In *Proceedings of the 2010 Workshop on Parallel Programming Patterns*. 1–9.

Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization.*

Tiantian Liu, Adam W Bargteil, James F O'Brien, and Ladislav Kavan. 2013. Fast simulation of mass-spring systems. *ACM Transactions on Graphics (TOG)* 32, 6 (2013), 1–7.

Miles Macklin, Matthias Müller, and Nuttapong Chentanez. 2016. XPBD: position-based simulation of compliant constrained dynamics. In *Proceedings of the 9th International Conference on Motion in Games*. 49–54.

Ahmed H. Mahmoud, Serban D. Porumbescu, and John D. Owens. 2021. RXMesh: A GPU Mesh Data Structure. *ACM Trans. Graph.* 40, 4, Article 104 (July 2021), 16 pages.

Martti Mäntylä. 1987. *An introduction to solid modeling.* Computer Science Press, Inc.

Nelson L. Max. 1999. Weights for Computing Vertex Normals from Facet Normals. *J. Graphics, GPU, & Game Tools* 4 (1999), 1–6.

Matthias Müller, Bruno Heidelberger, Marcus Hennix, and John Ratcliff. 2007. Position based dynamics. *Journal of Visual Communication and Image Representation* 18, 2 (2007), 109–118.

Ken Museth, Jeff Lait, John Johanson, Jeff Budsberg, Ron Henderson, Mihai Alden, Peter Cucka, David Hill, and Andrew Pearce. 2013. OpenVDB: an open-source data structure and toolkit for high-resolution volumes. In *Acm siggraph 2013 courses*. 1–1.

Rahul Narain, Matthew Overby, and George E Brown. 2016. ADMM⊇ projective dynamics: fast simulation of general constitutive models.. In *Symposium on Computer Animation*, Vol. 1. 2016.

Hang Si. 2015. TetGen, a Delaunay-Based Quality Tetrahedral Mesh Generator. *ACM Trans. Math. Softw.* 41, 2, Article 11 (feb 2015), 36 pages.

Eftychios Sifakis and Jernej Barbic. 2012. FEM simulation of 3D deformable solids: a practitioner's guide to theory, discretization and model reduction. In *Acm siggraph 2012 courses*. 1–50.

Demetri Terzopoulos, John Platt, Alan Barr, and Kurt Fleischer. 1987. Elastically deformable models. In *Proceedings of the 14th annual conference on Computer graphics and interactive techniques*. 205–214.

Zangyueyang Xian, Xin Tong, and Tiantian Liu. 2019. A scalable galerkin multigrid method for real-time simulation of deformable objects. *ACM Transactions on Graphics (TOG)* 38, 6 (2019), 1–13.

Cem Yuksel, Scott Schaefer, and John Keyser. 2009. Hair meshes. *ACM Transactions on Graphics (TOG)* 28, 5 (2009), 1–7.

Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. 2018. GraphIt: A high-performance graph DSL. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 121.

Yili Zhao and Jernej Barbič. 2013. Interactive authoring of simulation-ready plants. *ACM Transactions on Graphics (TOG)* 32, 4 (2013), 1–12.

Qingnan Zhou and Alec Jacobson. 2016. Thingi10K: A Dataset of 10,000 3D-Printing Models. *arXiv preprint arXiv:1605.04797* (2016).

A DETAILED PERFORMANCE FOR ALL EXAMPLES

We report the run time and compile time for different mesh-based tasks in Table 1 and Table 2. The run time measures the costs of mesh-related kernels to complete a geometry processing task (e.g. computing vertex normal) or to produce one time step for a physically-based simulation task (e.g. computing force in mass-spring system). As described in Section 5.1, our compiler partitions input meshes into patches and prepare the relations for each patch during compile time, we also cache these metadata into files. Our compiler will look for metadata first to see whether it needs to be cold-started. Our compiler re-computes all the metadata for changed meshes under cold-start settings. Otherwise, it reads the metadata from files directly. The cold-started and warm-started compile time columns report the compile time for these two situations respectively.

Table 1. Detailed Performance for the Mass-Spring Task in Figure 11

model	#verts.	#edges	#faces	#cells	compile time (cold-started)	compile time (warm-started)	mesh-kernel run time per time step (scattering)	(gathering)	(implicit)
bunny	1.1×10^4	3.1×10^4	2.1×10^4	-	0.91 sec	0.80 sec	0.007 ms	0.004 ms	0.308 ms
	2.1×10^4	6.2×10^4	4.1×10^4	-	0.94 sec	0.84 sec	0.008 ms	0.006 ms	0.341 ms
	4.1×10^4	1.2×10^5	8.3×10^4	-	1.11 sec	0.86 sec	0.013 ms	0.009 ms	0.477 ms
	8.3×10^4	2.5×10^5	1.7×10^5	-	1.49 sec	0.97 sec	0.019 ms	0.015 ms	0.583 ms
	1.7×10^5	5.1×10^5	3.4×10^5	-	2.29 sec	1.05 sec	0.040 ms	0.026 ms	0.912 ms
	3.4×10^5	1.0×10^6	6.8×10^5	-	4.24 sec	1.79 sec	0.083 ms	0.054 ms	1.671 ms
	6.9×10^5	2.1×10^6	1.4×10^6	-	8.46 sec	2.73 sec	0.164 ms	0.114 ms	3.408 ms
	1.4×10^6	4.2×10^6	2.8×10^6	-	17.18 sec	4.46 sec	0.323 ms	0.229 ms	6.721 ms
	2.9×10^6	8.6×10^6	5.7×10^6	-	35.76 sec	8.10 sec	0.666 ms	0.472 ms	13.142 ms
	5.8×10^6	1.7×10^7	1.2×10^7	-	75.30 sec	15.92 sec	1.290 ms	0.929 ms	27.104 ms

Table 2. Detailed Performance for Other Tasks

task	model	#verts.	#edges	#faces	#cells	compile time (cold-started)	compile time (warm-started)	run time (mesh-kernels)
geodesic distance	buddha	4.1×10^6	1.2×10^7	8.1×10^6	-	44.54 sec	4.12 sec	1.015 sec
	goyle	6.3×10^6	1.9×10^7	1.3×10^7	-	64.49 sec	6.26 sec	1.376 sec
	lucy	2.0×10^7	6.1×10^7	4.0×10^7	-	239.57 sec	19.53 sec	3.293 sec
	venus	2.5×10^7	7.6×10^7	5.0×10^7	-	359.43 sec	91.43 sec	5.356 sec
	vlamp	9.6×10^6	2.9×10^7	1.9×10^7	-	129.64 sec	9.72 sec	1.800 sec
vertex normal	buddha	4.1×10^6	1.2×10^7	8.1×10^6	-	44.54 sec	4.12 sec	0.713 ms
	fluid	3.7×10^6	1.1×10^7	7.5×10^6	-	39.58 sec	4.64 sec	0.431 ms
	goyle	6.3×10^6	1.9×10^7	1.3×10^7	-	64.49 sec	6.26 sec	0.770 ms
	lucy	2.0×10^7	6.1×10^7	4.0×10^7	-	239.57 sec	19.53 sec	3.891 ms
	venus	2.5×10^7	7.6×10^7	5.0×10^7	-	359.43 sec	91.43 sec	3.977 ms
	vlamp	9.6×10^6	2.9×10^7	1.9×10^7	-	129.64 sec	9.72 sec	1.920 ms
projective dynamics	deer	1.9×10^5	3.4×10^5	3.0×10^5	7.0×10^5	18.40 sec	1.72 sec	29.411 ms
XPBD cloth sim	cloth	1.0×10^6	3.0×10^6	2.0×10^6	-	9.26 sec	6.05	4.192 ms
mesh-grid hybrid sim	armadillo	2.2×10^8	5.7×10^8	3.8×10^8	7.1×10^8	208.30 [a] sec	19.05 sec	0.063 sec

[a]In the mesh-grid hybrid simulation, we group 270 armadillos in a batch and repeat the patching information of those 270 armadillos for 63 times.

B ESTIMATED MEMORY FOOTPRINT FOR RELATIONS

Our compiler pre-computes the relations in compile time. Apparently we do not want to compute and store every pair of relations of input meshes. We, therefore, compute only the wanted relations by inspecting user's `mesh-for` kernels. In this section, we report our estimated memory footprint for relations.

Let us consider a well-tessellated surface manifold mesh whose number of vertices, edges and faces are approximately $1 : 3 : 2$. And we assume the each index in a relation is stored in a two-byte unsigned integer for patches. Under these assumptions, the linear algebraic representation (LAR) [DiCarlo et al. 2014] uses $12n_{\mathcal{F}}$ to store all $\mathcal{FE}$ and $\mathcal{EV}$ relations regardless of tasks, where $n_{\mathcal{F}}$ is the number of faces of a certain mesh. Unlike the LAR, our compiler only stores the wanted relations, so the memory footprint is task dependent. For example, the surface normal computation requires only an $\mathcal{FV}$ relation which will take $6n_{\mathcal{F}}$ bytes; the geodesic distance computation involves both $\mathcal{VF}$ and $\mathcal{FV}$ relations which takes $13n_{\mathcal{F}}$ bytes; the scattering-style spring force computation uses only an $\mathcal{EV}$ relation that takes $6n_{\mathcal{F}}$ bytes; and the gathering-style spring force computation uses an $\mathcal{VV}$ relation and takes $7n_{\mathcal{F}}$ bytes. In complex cases such as the cloth simulation using XPBD that requires all $\mathcal{EV}$, $\mathcal{EF}$ and $\mathcal{FV}$ relations, it takes us $21n_{\mathcal{F}}$ bytes to store all the relations. As we can see, our compiler may consume larger memory compared with LAR for complex relation accesses. But we do not need to compute relations in run-time as the return of our memory investment. Note that we use ribbons to pad each patch, so the actual memory footprint is slightly larger than our estimations for both LAR and our relation representation.